```
// Last updated 2021-04-29 09:00
#time "on"

#r "nuget: FSharp.Data"
#r "nuget: FSharp.Stats"
#r "nuget: NodaTime"

#load "Portfolio.fsx"

open System
open System.IO
open System.IO.Compression
open FSharp.Data
open NodaTime
open NodaTime.Calendars
open FSharp.Stats

open Portfolio

Environment.CurrentDirectory <- __SOURCE_DIRECTORY__
fsi.AddPrinter<DateTime>(fun dt -> dt.ToString("s"))
fsi.AddPrinter<YearMonth>(fun ym -> $"{ym.Year}-%02i{ym.Month}")

let samplePeriod x =
    x >= YearMonth(2010, 1) &&
    x <= YearMonth(2020, 2)
```

Price momentum

Price momentum is one of the most common quant signals. It is (fairly) straight forward to calculate, and you only need returns to do it, so it is a good starting point and reference 'strategy'.

```
// Now create a type that represents the file.
// This figures out what the columns of the file are.
//  - Sample is the path to our file. The "../" means we're
//    doing relative paths, so we need to specify the
//  - ResolutionFolder to indicate what folder the relative paths
//    are relative to.
type MsfCsv = CsvProvider<Sample="../data-cache/msf-momentum2.csv",
                          ResolutionFolder = __SOURCE_DIRECTORY__>

// assign the content of the file to a value
let msfCsv = MsfCsv.GetSample()

// look at the file attributes
msfCsv
// look at the headers
msfCsv.Headers
// look at the first few rows
msfCsv.Rows |> Seq.truncate 3
```

Signal construction

We want to create a momentum signal and see how it relates to future returns. The signal is some measure of past returns. A common measure is the past year return, skipping the most recent month. We skip the most recent month because stocks tend to reverse following very recent returns (known as "reversals"). The reversal is very likely a liquidity effect and it is less important this century. So returns are positively correlated with returns from 12 months to 1 months ago, but negatively correlated with returns last month. This is illustrated very nicely in Jegadeesh (1990).

If we're forming a portfolio at the end of month $t-1$ to hold in month t, then we're saying returns in month t are positively correlated with returns in months $t-12$ through month $t-2$. For example, if we want to hold a momenum portfolio in January 2021, then we will form it on December 31, 2020. We will want to go long stocks that had high returns from the beginning of January 2020 to the end of November 2020.

Let's create a record to hold some info about past returns for a stock. We will use this as a trading signal.

```
type PriorReturnOb =
    { SecurityId : SecurityId
      FormationMonth : YearMonth
      Retm12m2 : float
      N : int }
```

Note the `YearMonth` type for portfolio formation month. This type is from the library NodaTime.

Why are we using it? Date math is hard and easy to mess up.

We're dealing with monthly data. If we use `DateTime`, then we have to give the month a day value. We could always use the first day of the month, but then month return goes all the way to the end of the month. And we might forget that information.

If we use the last day of the month, then what happens when we add months. For example, we have to start doing things like.

```
let endOfFebruary = DateTime(2020,2,28)
let endOfFebruaryPlus1Month = endOfFebruary.AddMonths(1)
let endOfMarch = DateTime(endOfFebruary.Year,endOfFebruary.Month,1).AddMonths(2).AddDays(-1.0)
endOfFebruaryPlus1Month = endOfMarch // evaluates to false
```

That's kind of ugly.

We also have to worry about things like what happens if we add a month but it overlapped with daylight savings time? What about timezones? If we're never dealing with times, it's nice to ignore all these things.

This is nicer way of doing it using nodatime's `YearMonth`:

```
let february = YearMonth(2020,2)
let februaryPlus1Month = february.PlusMonths(1)
let march = YearMonth(2020,3)
februaryPlus1Month = march // true
```

Let's focus on a single stock.

```
let amznReturns =
    // we're filtering and then storing as a map.
    // if we used a sequence instead of a map/arra/list, then
    // every time we used amznReturns, the sequence
    // would be recreated by filtering msfCsv.Rows.
    // That's one difference between (lazy) seq and
    // (eager) array/list/map.
    msfCsv.Rows
    |> Seq.filter(fun x -> x.Ticker = "AMZN")
    |> Seq.map(fun x ->
        let ym = YearMonth(x.Month.Year,x.Month.Month)
        let key = Permno x.Permno, ym
        key, x)
    |> Map.ofSeq

let getPastYearObs
    (returns:Map<(SecurityId * YearMonth),MsfCsv.Row>)
    (security: SecurityId, formationMonth: YearMonth) =
        [| -11 .. -1 |]
        |> Array.choose(fun i ->
            let returnMonth = formationMonth.PlusMonths(i)
            Map.tryFind (security, returnMonth) returns)

// check Permno 84788 is Amzn
let amznPermno = Permno 84788
getPastYearObs amznReturns (amznPermno, YearMonth(2019,1))
getPastYearObs amznReturns (Permno -400, YearMonth(2019,1))

// making cumulative returns
let cumulativeReturn rets =
    // using Seq so that it will work with any collection
    let grossReturn = (1.0, rets) ||> Seq.fold(fun acc ret -> acc * (1.0 + ret))
    grossReturn - 1.0

// check
cumulativeReturn []
cumulativeReturn [1.0;-0.5]
```

We're now ready to create our Momentum signal function.

```
// If you don't want to write the typessecurity, month all the time.
//

let getMomentumSignal returns (security, formationMonth) =
    let priorObs = getPastYearObs  returns (security, formationMonth)
    let priorRets = priorObs |> Array.choose(fun x -> x.Ret)
    // We should probably return None if there are no observations.
    // If they are all missing, Array.choose will return an empty
    // array. See:
    // ([| None; None |]: int option []) |> Array.choose id
    //
    // So we'll check for an empty array and return None in that case.
    if Array.isEmpty priorRets then
        None
    else
        Some { SecurityId = security
               FormationMonth = formationMonth
               Retm12m2 = cumulativeReturn priorRets
               N = priorRets.Length }

// Check
getMomentumSignal amznReturns (amznPermno, YearMonth(2019,1))
getMomentumSignal amznReturns (Permno -400, YearMonth(2019,1))
```

One thing you may notice is that our momentum signal function gets everything from it's inputs. That means that if we give it different intputs then we could get momentum signals for other stocks.

For example we can create a map collection like we had for amzn, but for all stocks.

```
let msfByPermnoMonth =
    msfCsv.Rows
    |> Seq.map(fun x ->
        let ym = YearMonth(x.Month.Year,x.Month.Month)
        let key = Permno x.Permno, ym
        key, x)
    |> Map.ofSeq

// finding some permnos for notable tickers

// don't use tickers. companies change tickers, so you might look up the wrong company
// That's why I'm picking some tickers that I know haven't changed, but my function
// is using PERMNO.

let notableTicks =
    ["MSFT";"AAPL";"HOG"]
    |> Seq.map(fun tick ->
        msfCsv.Rows
        // hover over 'find' in 'Seq.find' if you don't remember what it does.
        |> Seq.find(fun row -> row.Ticker = tick)
        |> fun row -> row.Ticker, row.Permno)
    |> Map.ofSeq

let msftPermno = Permno notableTicks.["MSFT"]
let aaplPermno = Permno notableTicks.["AAPL"]
let hogPermno = Permno notableTicks.["HOG"]

let msftTestIndex = (msftPermno, YearMonth(2019,1))
let aaplTestIndex = (aaplPermno, YearMonth(2019,1))

getMomentumSignal msfByPermnoMonth msftTestIndex
getMomentumSignal msfByPermnoMonth aaplTestIndex

(*
and we can use [partial function application](https://fsharpforfunandprofit.com/posts/partial-application/)
to "bake in" the msfByPermnoMonth parameter so that we don't keep having to pass it around.
*)

let getMomentumSignalAny = getMomentumSignal msfByPermnoMonth

getMomentumSignalAny msftTestIndex
getMomentumSignalAny aaplTestIndex
```

Defining the investment universe

Let's say we have a portfolio formation month. Can we look up securities available to invest in?

```
let securitiesByFormationMonth =
    msfCsv.Rows
    |> Seq.groupBy(fun x -> YearMonth(x.Month.Year, x.Month.Month))
    |> Seq.map(fun (ym, xs) ->
        ym,
        xs
        |> Seq.map(fun x -> Permno x.Permno)
        |> Seq.toArray)
    |> Map.ofSeq

let getInvestmentUniverse formationMonth =
    match Map.tryFind formationMonth securitiesByFormationMonth with
    | Some securities ->
        { FormationMonth = formationMonth
          Securities = securities }
    | None -> failwith $"{formationMonth} is not in the date range"

getInvestmentUniverse (YearMonth(2011,10))
// getInvestmentUniverse (YearMonth(1990,10))
```

You might also want to filter the investment universe by some criteria.

```
let isCommonStock securityFormationMonth =
    match Map.tryFind securityFormationMonth msfByPermnoMonth with
    | None -> false
    | Some x -> List.contains x.Shrcd [10; 11]

let onNyseNasdaqAmex securityFormationMonth =
    match Map.tryFind securityFormationMonth msfByPermnoMonth with
    | None -> false
    | Some x -> List.contains x.Exchcd [ 1; 2; 3]

let hasPrice13mAgo (security, formationMonth:YearMonth) =
    //13m before the holding month, 12m before the formation month
    match Map.tryFind (security, formationMonth.PlusMonths(-12)) msfByPermnoMonth with
    | None -> false
    | Some m13 -> m13.Prc.IsSome

let hasReturn2mAgo (security, formationMonth:YearMonth) =
    //2m before the holding month, 1m before the formation month
    match Map.tryFind (security, formationMonth.PlusMonths(-1)) msfByPermnoMonth with
    | None -> false
    | Some m2 -> m2.Ret.IsSome

let hasMe1mAgo (security, formationMonth) =
    //1m before the holding month, so the formation month
    match Map.tryFind (security, formationMonth) msfByPermnoMonth with
    | None -> false
    | Some m1 -> m1.Prc.IsSome && m1.Shrout.IsSome

let has8ReturnsPastYear securityFormationMonth =
    match getMomentumSignalAny securityFormationMonth with
    | None -> false
    | Some x -> x.N >= 8

let danielMoskowitzRestrictions securityFormationMonth =
    isCommonStock securityFormationMonth &&
    onNyseNasdaqAmex securityFormationMonth &&
    hasPrice13mAgo securityFormationMonth &&
    hasReturn2mAgo securityFormationMonth &&
    hasMe1mAgo securityFormationMonth &&
    has8ReturnsPastYear securityFormationMonth

let restrictUniverse (investmentUniverse: InvestmentUniverse) =
    let filtered =
        investmentUniverse.Securities
        |> Array.filter(fun security ->
            danielMoskowitzRestrictions (security, investmentUniverse.FormationMonth))
    { FormationMonth = investmentUniverse.FormationMonth
      Securities = filtered }
```

Now we can see where we are.

```
YearMonth(2011,10)
|> getInvestmentUniverse
|> restrictUniverse

let investmentUniverse =
    YearMonth(2011,10)
    |> getInvestmentUniverse
let restrictedInvestmentUniverse =
    investmentUniverse |> restrictUniverse

// See if we're excluding some securities.
investmentUniverse.Securities.Length
restrictedInvestmentUniverse.Securities.Length
```

Momentum signals for our investment universe

Let's look at how to transform our array of securities in our investment universe into an array with the signals.

Recall that our momentum function returns a type of observation specific to momentum.

```
getMomentumSignalAny (Permno 84788, YearMonth(2019,1))
```

```
Real: 00:00:00.000, CPU: 00:00:00.000, GC gen0: 0, gen1: 0, gen2: 0
val it : PriorReturnOb option = Some { SecurityId = Permno 84788
                                       FormationMonth = 2019-01
                                       Retm12m2 = 0.03520594654
                                       N = 11 }
```

This is fine, but if we want our code to work with any type of signal, then we need to transform it into something more generic.

This is the purpose of the `SecuritySignal` record in the `Portfolio` module. It's the same thing that we had in the simpler portfolio formation example. It can represent any signal that is a float. And to hold an array of security signals for a particular month, we now have the type `SecuritiesWithSignals` also from the `Portfolio` module.

Let's write a function that transforms our momentum signal into a more generic security signal.

```
let getMomentumSecuritySignal (security, formationMonth ) =
    match getMomentumSignalAny (security, formationMonth) with
    | None -> None
    | Some signalOb ->
        let signal = { SecurityId = security; Signal = signalOb.Retm12m2 }
        Some signal
```

Now compare

```
getMomentumSignalAny (Permno 84788, YearMonth(2019,1))
```

```
Real: 00:00:00.000, CPU: 00:00:00.000, GC gen0: 0, gen1: 0, gen2: 0
val it : PriorReturnOb option = Some { SecurityId = Permno 84788
                                       FormationMonth = 2019-01
                                       Retm12m2 = 0.03520594654
                                       N = 11 }
```

```
getMomentumSecuritySignal (Permno 84788, YearMonth(2019,1))
```

```
Real: 00:00:00.000, CPU: 00:00:00.000, GC gen0: 0, gen1: 0, gen2: 0
val it : SecuritySignal option = Some { SecurityId = Permno 84788
                                        Signal = 0.03520594654 }
```

Now a function that takes our investment universe and returns our securities with their (now more generic) signal.

```
let getMomentumSignals (investmentUniverse: InvestmentUniverse) =
    let arrayOfSecuritySignals =
        investmentUniverse.Securities
        |> Array.choose(fun security ->
            getMomentumSecuritySignal (security, investmentUniverse.FormationMonth))

    { FormationMonth = investmentUniverse.FormationMonth
      Signals = arrayOfSecuritySignals }

restrictedInvestmentUniverse
|> getMomentumSignals


// or, if we want to look at the full pipeline
YearMonth(2015,7)
|> getInvestmentUniverse
|> restrictUniverse
|> getMomentumSignals
```

Assigning portfolios

Now that we have the signals for our portfolio, we can assign portfolios. For many strategies it is common to use decile sorts. This means that you sort securities into 10 portfolios based on the signal. But other numbers of portfolios (tercile = 3, quintile = 5, etc) are also common.

There's a tradeoff between signal strength and diversification. More portfolios means that top/bottom portfolios are stronger bets on the signal. But there are fewer securities, so they are also less diversified. Often, the trade-off between a stronger signal vs. less diversificaton balances out. Specifically, long-short tercile sorts may have a lower return spread than decile sorts. But since terciles are better diversified, the tercile and decile shorts are not as different when looking at sharpe ratios.

```
let assignSignalSorts name n { FormationMonth = ym; Signals = xs } =
    xs
    |> Array.sortBy(fun x -> x.Signal)
    |> Array.splitInto n
    |> Array.mapi(fun i ys ->
        // because arrays are 0-indexed and I want the minimum
        // portfolio index to be 1, I'm doing index = i+1.
        { PortfolioId = Indexed(name=name,index=i+1)
          Signals = ys })
```

```
YearMonth(2015,7)
|> getInvestmentUniverse
|> restrictUniverse
|> getMomentumSignals
|> assignSignalSort "Momentum" 10
```

Calculating Portfolio weights

We'll use a value-weight scheme. So we need a function that gets market capitalizations.

```
let getMarketCap (security, formationMonth) =
    match Map.tryFind (security, formationMonth) msfByPermnoMonth with
    | None -> None
    | Some x ->
        match x.Prc, x.Shrout with
        // we need valid price and valid shrout to get market caps
        | Some prc, Some shrout -> Some (security, prc*shrout)
        // anything else and we can't calculate it
        | _ -> None
```

Some examples.

```
getMarketCap (amznPermno, YearMonth(2019,1)) // Some marketCap
getMarketCap (amznPermno, YearMonth(2030,1)) // In the future, so None

let exampleCapitalizations =
    [| getMarketCap (amznPermno, YearMonth(2019,1))
       getMarketCap (hogPermno, YearMonth(2019,1)) |]
    |> Array.choose id // to unwrap the options

let exampleValueWeights =
    let tot = exampleCapitalizations |> Array.sumBy snd
    exampleCapitalizations
    |> Array.map(fun (_id, cap) -> cap / tot )
    |> Array.sortDescending
```

```
Real: 00:00:00.001, CPU: 00:00:00.000, GC gen0: 0, gen1: 0, gen2: 0
val exampleCapitalizations : (SecurityId * float) [] =
  [|(Permno 84788, 844245322.6); (Permno 70033, 6002024.479)|]
val exampleValueWeights : float [] = [|0.9929408489; 0.007059151081|]
```

Now imagining we have the same example in terms of an assigned portfolio with made up signals.

```
let mktCapExPort: AssignedPortfolio =
    { PortfolioId = Named("Mkt Cap Example")
      FormationMonth = YearMonth(2019,1)
      Signals = [|{ SecurityId = amznPermno; Signal = 1.0 }
                  { SecurityId = hogPermno; Signal = 1.0 } |] }
```

The portfolio module has a function that can help us. It has two inputs.

- A function that gets market capitalizations
- An assigned portfolio

We should see that it gives the same value weights.

```
let exampleValueWeights2 =
    giveValueWeights getMarketCap mktCapExPort
```

```
Real: 00:00:00.001, CPU: 00:00:00.000, GC gen0: 0, gen1: 0, gen2: 0
val exampleValueWeights2 : Portfolio =
  { PortfolioId = Named "Mkt Cap Example"
    FormationMonth = 2019-01
    Positions = [|{ SecurityId = Permno 84788
                    Weight = 0.9929408489 }; { SecurityId = Permno 70033
                                               Weight = 0.007059151081 }|] }
```

So now we can construct our portfolios with value weights.

```
let portfoliosWithWeights =
    YearMonth(2015,7)
    |> getInvestmentUniverse
    |> restrictUniverse
    |> getMomentumSignals
    |> assignSignalSort "Momentum" 10
    |> Array.map (giveValueWeights getMarketCap)
```

Note that because of the size distribution, some of these portfolios are not very diversified. This is illustrated by inspecting maximum portfolio weights.

```
portfoliosWithWeights
|> Array.map(fun port ->
    port.PortfolioId,
    port.Positions |> Array.map(fun pos -> pos.Weight) |> Array.max,
    port.Positions |> Array.sumBy(fun pos -> pos.Weight))
```

Calculating Portfolio returns

We need our function to get returns given weights.

We can start with a function that gets returns. It looks a lot like our function to get market capitalizations.

```
let getSecurityReturn (security, formationMonth) =
    // If the security has a missing return, assume that we got 0.0.
    // Note: If we were doing excess returns, we'd need 0.0 - rf
    let missingReturn = 0.0
    match Map.tryFind (security, formationMonth) msfByPermnoMonth with
    | None -> security, missingReturn
    | Some x ->
        match x.Ret with
        | None -> security, missingReturn
        | Some r -> security, r

getSecurityReturn (amznPermno, YearMonth(2019,1))

let portReturn =
    getPortfolioReturn getSecurityReturn exampleValueWeights2

portfoliosWithWeights
|> Array.map (getPortfolioReturn getSecurityReturn)


// Put it all together.
let sampleMonths = getSampleMonths (YearMonth(2010,5), YearMonth(2020,2))

sampleMonths |> List.rev |> List.take 3

let formMomtenumPort ym =
    ym
    |> getInvestmentUniverse
    |> restrictUniverse
    |> getMomentumSignals
    |> assignSignalSort "Momentum" 10
    |> Array.map (giveValueWeights getMarketCap)
    |> Array.map (getPortfolioReturn getSecurityReturn)
```

We can process months sequentially.

```
let momentumPortsSequential =
    sampleMonths
    |> List.toArray
    |> Array.collect formMomtenumPort
```

Or we can speed things up and process months in Parallel using all available CPU cores. Note that the only change is that we use `Array.Parallel.collect` intead of `Array.collect`. The array collection is the only parallel collection in base F# and the module functions are somewhat limited, but if you google F# parallel processing you can find other options and also asynchrounous coding.

```
let momentumPortsParallel =
    sampleMonths
    |> List.toArray
    |> Array.Parallel.collect formMomtenumPort
```